**《Effective C#》：值类型和引用类型**

http://www.sina.com.cn 2007年08月16日 08:24  [天极yesky](http://www.yesky.com/)

　　作者： 愚翁 出处： 天极Yesky软件频道

　　在C#中有两种类型的数据，一种是值类型数据，一种是引用类型数据。在编码的时候区分这两种类型数据，可以避免一些细小的编码错误。   
  
　 　首先说说什么类型是值类型，例如：int、float、bool之类的基础类型，以及用struct定义的类型，如：DateTime。除此外，如 string，数组，以及用class定义的类型等都是引用类型。对于C#来说，很难罗列出所有类型进行一一分别，这需要自己在编码过程中进行分析总结。  
  
　　为了更好地说明两种类型之间的区别，借用如下的表格来说明。

|  |  |  |
| --- | --- | --- |
|  | 值类型 | 引用类型 |
| 内存分配地点 | 分配在栈中 | 分配在堆中 |
| 效率 | 效率高，不需要地址转换 | 效率低，需要进行地址转换 |
| 内存回收 | 使用完后，立即回收 | 使用完后，不是立即回收，等待GC回收 |
| 赋值操作 | 进行复制，创建一个同值新对象 | 只是对原有对象的引用 |
| 函数参数与返回值 | 是对象的复制 | 是原有对象的引用，并不产生新的对象 |
| 类型扩展 | 不易扩展 | 容易扩展，方便与类型扩展 |

　　通过如上细致对比，大家对于值类型和引用类型有个清楚的概念。  
  
　　不过，无论是对于值类型还是引用类型来说，对于其作为函数参数或者返回值的时候，都是容易犯错误的地方。  
  
　　对于值类型来说，当其作为函数参数的时候，希望在函数中被修改，那么直接如下操作是不能被修改的。

|  |
| --- |
| public void Increment( int i ) { 　 i++; } |

　　要想在函数中对传进去的参数做真正的修改，需要借助于**ref**这个关键字，那么正确的形式如下。

|  |
| --- |
| public void Increment( **ref** int i ) { 　i++; } |

　　也就是说，如果需要在函数中对值类型参数进行修改，需要用ref或者out进行标识才能真正实现。  
  
　　而对于引用类型来说，当其作为函数参数的时候，它所遇到的情况恰恰与值类型相反，即不希望在函数中被修改，举例如下。

|  |
| --- |
| public void AddValue( MyType typValue ) { 　typValue.Count = typValue.Count + 15; } |

　　由于对于引用类型对象来说，其的赋值操作只是对原有对象的引用，因此在函数对其修改，实际上是直接修改了原有对象数据，这是很多情况不希望发生的（这里例如对数组或者DataTable操作这类）。  
  
　　为了防止这种事发生，需要给此类型提供clone函数。例如对于如上的类型，可以入下实现。

|  |
| --- |
| public class MyType:ICloneable { 　private int nCount = 0; 　public int Count 　{ 　　set{ nCount = value;} 　　get{ return nCount;} 　} 　public MyType() 　{} 　public MyType( int Value) 　{ 　　nCount = Value; 　}  　#region ICloneable Members  　public object Clone() 　{ 　　// TODO: Add MyType.Clone implementation 　　return new MyType( nCount ); 　}  　#endregion } |

　　那么在调用的时候，用当前的对象的clone作为参数即可。  
  
　 　不过对于引用类型来说，提供一个clone函数不是一件容易的事情，尤其出现引用类型嵌套的时候，所以说去实现一个完全clone功能是件很费事又不讨 好的活，这也就是在论坛中常说的深copy和浅copy的问题。话虽如此，如果对于前面所说的有个大概了解，相信实现也不是不可能。  
  
　　在C#中，尤其自己定义类型的时候，常常由于是用struct来定义还是用class来定义，即是定义一个值类型还是一个引用类型呢。在这本书上给了几个判定条件，如果如下几点都满足的话，建议用struct来定义为值类型，否则用class定义为引用类型。

|  |
| --- |
| <!--[if !supportLists]-->1． <!--[endif]-->这个类型是否主要为了数据存储； <!--[if !supportLists]-->2． <!--[endif]-->是否只通过属性来访问对象的数据成员； <!--[if !supportLists]-->3． <!--[endif]-->这个类型是否不会有子类型； <!--[if !supportLists]-->4． <!--[endif]-->在程序处理的时候不会把这个类型对象通过多态来处理。 |

楼主你好，这是个Lambda表达式。

Lambda表达式是C#3.0的新内容，如果您之前学习的C#2.0，不认识也就不奇怪了。

给您举个例子。

例如，我定义一个委托：

**delegate** int Method(int a, int b);

再定义一个方法：

int Add(int a, int b)

{

return a + b;

}

我可能需要这样通过委托调用方法：

**Method m += Add;**

**Console.WriteLine(m(2, 3));**

====================================================

但如果用 Lambda 表达式：

Method m += (a ,b) => a + b;

Console.WriteLine(m(2, 3));

可以省去方法的定义。

实际上， Lambda 表达式只是简化了匿名方法的语法而已。

附 C# 2.0 的匿名方法语法：

Method m += delegate(int a, int b) { return a + b; };

Console.WriteLine(m(2, 3));